Arquitetura de Computadores

Paralelismo em Nível de Instrução

Paralelismo em Nível de Instrução

- Todos os processadores a partir de 1985 usam pipelining para sobrepor a execução de instruções e melhorar seu desempenho.
- Esse potencial de sobreposição de instruções é chamado de Paralelismo em Nível de Instruções, **ILP** (do inglês- ***Instruction Level Parallelism***).

Bloco Básico

- Um bloco básico é uma sequência de código em linha reta sem ramificações, exceto para a entrada e saída.
- A quantidade de paralelismo disponível dentro de um bloco básico é bem pequena.
- Para programas RISC típicos, a frequência média de desvio dinâmico geralmente está entre 15% e 25%, o que significa que entre três e seis instruções são executadas entre um par de desvios, geralmente essas instruções dependem uma das outras o que não permite paralelismo.
- Para obter melhorias substanciais de desempenho, devemos explorar o ILP em vários blocos básicos.

Paralelismo em Nível de Laço

- A maneira mais simples e comum de aumentar o ILP é explorar o paralelismo entre as iterações de um laço(loop). Esse tipo de paralelismo é frequentemente chamado de **paralelismo em nível de laço**.
- Cada iteração do laço pode se sobrepor a qualquer outra iteração, embora dentro de cada iteração do laço haja pouca ou nenhuma oportunidade de sobreposição.

Paralelismo em Nível de Instrução

- Examinaremos várias técnicas para converter esse paralelismo em nível de laço em paralelismo em nível de instrução.
- Basicamente, essas técnicas funcionam “desenrolando” o laço estaticamente pelo compilador ou dinamicamente pelo hardware.

Dependência de Dados

- Existe um problema ao explorar o ILP, a dependência de dados.
- As dependências são uma propriedade dos programas.
- A organização do pipeline determina se a dependência é detectada e se causa uma paralisação(stall).
- Dependências que fluem através de locais de memória são difíceis de detectar.

Dependência de Dados

- Existem três tipos diferentes de dependências:
  - dependências de dados (também chamadas de dependências de dados verdadeiras),
  - dependências de nome; e
  - dependências de controle.
- Uma instrução j é dependente de dados da instrução i se qualquer um dos seguintes ocorrer:
  - A instrução i produz um resultado que pode ser usado pela instrução j.
  - A instrução j é dependente de dados da instrução k, e a instrução k é dependente de dados da instrução i.

Dependência de Dados

- Por exemplo, considere a seguinte sequência de código RISC-V que incrementa um vetor de valores na memória (começando em 0(x1) terminando com o último elemento em 0(x2)) por um escalar no registrador f2.

```
Loop: fld f0,0(x1)        //carrega o 1º elemento do vetor em f0
      fadd.d f4,f0,f2     //soma o escalar em f2 a f0 e guarda em f4
      fsd f4,0(x1)        //armazena o resultado de volta no vetor
      addi x1,x1,-8       //decrementa o ponteiro em 8 bytes – próx.vetor
      bne x1,x2,Loop      //salta, se não igual x1 e x2, para loop
```

Dependência de Dados

- As dependências de dados nesta sequência de código envolvem dados de ponto flutuante:

```
Loop: fld f0,0(x1)
      fadd.d f4,f0,f2     // esta instrução depende da anterior - f0
      fsd f4,0(x1)        // esta instrução depende da anterior - f4
```

- e de dados inteiros:

```
      addi x1,x1,-8
      bne x1,x2,Loop  // esta instrução depende da anterior - x1
```

Dependência de Dados

- Quando uma instrução é dependente de dados de outra é necessário que a ordem de execução das instruções sejam mantidas, logo não podem ser executadas simultaneamente ou serem completamente sobrepostas.
- Uma dependência de dados determina três coisas:

  (1) a possibilidade de um conflito,

  (2) a ordem em que os resultados devem ser calculados e

  (3) um limite superior de quanto paralelismo pode ser explorado.

Dependência de Dados

- Uma dependência de dados limita a quantidade de paralelismo em nível de instrução que podemos explorar.
- Uma dependência pode ser superada de duas maneiras diferentes:

  (1) mantendo a dependência, mas evitando um conflito, e

  (2) eliminando uma dependência transformando o código

Escalonamento de Código

- O **escalonamento do código** é o principal método usado para evitar um conflito sem alterar uma dependência, e esse escalonamento pode ser feito tanto pelo compilador quanto pelo hardware.
- Um valor de dado pode fluir entre instruções por meio de registradores ou por locais de memória. Quando o fluxo de dados ocorre através de um registrador, a detecção da dependência é direta porque os nomes dos registradores são fixados nas instruções, embora fique mais complicado quando os ramos intervêm e as questões de correção forçam um compilador ou hardware a ser conservador.
- As dependências que fluem pelos locais de memória são mais difíceis de detectar porque dois endereços podem se referir ao mesmo local, mas parecem diferentes: Por exemplo, 100(x4) e 20(x6) podem ser endereços de memória idênticos.

Dependência de Dados

- Dependência de Nomes
  - Uma dependência de nome ocorre quando duas instruções usam o mesmo registrador ou local de memória, chamado de nome, mas não há fluxo de dados entre as instruções associadas a esse nome. Existem dois tipos de dependências de nomes entre uma instrução i que precede a instrução j na ordem do programa:
    - Uma **antidependência** entre a instrução i e a instrução j ocorre quando a instrução j escreve um registrador ou local de memória que a instrução i lê. A ordenação original deve ser preservada para garantir que i leia o valor correto
    - Uma **dependência de saída** ocorre quando a instrução i e a instrução j escrevem no mesmo registrador ou local de memória. A ordenação entre as instruções deve ser preservada para garantir que o valor finalmente escrito corresponda à instrução j.

Dependência de Dados

- Antidependências e dependências de saída são dependências de nome, ao contrário de dependências de dados verdadeiras, porque não há valor sendo transmitido entre as instruções.
- Logo podem ser executadas simultaneamente ou em ordem trocada se o nome usado for trocado para evitar conflitos.

Ordem do Programa

- Conflito de Dados
  - Existe um conflito sempre que há uma dependência de nome ou dados entre as instruções, e elas estão próximas o suficiente para que a sobreposição durante a execução altere a ordem de acesso ao operando envolvido na dependência.
  - Por causa da dependência, devemos preservar o que é chamado de **ordem do programa** - isto é, a ordem em que as instruções seriam executadas se executadas sequencialmente uma de cada vez, conforme determinado pelo programa fonte original.

Conflito de Dados

- Os conflitos de dados podem ser classificados em três tipos, dependendo da ordem dos acessos de leitura e gravação nas instruções:
- Considere duas instruções i e j, com i precedendo j na ordem do programa. Os possíveis riscos de dados são:
  - RAW (read after write): j tenta ler uma fonte antes de i escrevê-la, então j incorretamente obtém o valor antigo. Este conflito é o tipo mais comum e corresponde a uma verdadeira dependência de dados. A ordem do programa deve ser preservada para garantir que j receba o valor de i.

Conflito de Dados

- WAW (write after write)—j tenta escrever um operando antes de ser escrito por i. As escritas acabam sendo realizadas na ordem errada, deixando o valor escrito por i ao invés do valor escrito por j no destino. Este perigo corresponde a uma dependência de saída.
- WAR (write after read)—j tenta escrever um destino antes de ser lido por i, então i obtém incorretamente o novo valor. Este perigo surge de uma antidependência (ou dependência do nome).

Dependência de Controle

- Determina a ordenação de uma instrução i com relação a uma instrução de desvio, de modo que essa instrução seja executada na ordem correta do programa e somente quando precisar.
- Cada instrução, fora do primeiro bloco básico do programa, é dependente de controle em algum conjunto de desvios.

Dependência de Controle

- Por exemplo, no segmento de código:

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

- S1 é dependente de controle de p1, e
- S2 é dependente de controle de p2, mas não de p1.

Dependência de Controle

- 1. Uma instrução que é dependente de controle em um desvio não pode ser movida antes do desvio;
  - por exemplo, não podemos pegar uma instrução da parte then de uma instrução if e movê-la antes da instrução if.
- 2. Uma instrução que não é dependente de controle em um desvio não pode ser movida para depois do desvio.
  - por exemplo, não podemos pegar uma instrução antes da instrução if e movê-la para a parte then.

Exercícios

- Para o seguinte trecho de código

```
loop:  ld  x1, 64(x6)
       add x2, x1, x3
       sub x2, x4, x5
       or   x4, x5, x6
       add x6, x6, -8
       bne x6, x7, loop
```

  - Determine os tipos de dependências que ocorrem.
  - Desenrole o laço em um fator de 4.

Técnicas Básicas para Expor ILP

- Desdobramento de laço e reescalonamento de código
  - Seja o laço:

    ```
    for (i=999; i>=0; i--)
       x[i] = x[i] + s;
    ```

  - Traduzindo fica:

    ```
    _loop : ld  f0, 0(x1)  // f0=elemento do vetor
            add f4, f0, f2 // f2 contem o valor de s
            sw f4, 0(x1)  // armazena o resultado
            add x1, x1, -8          // x1=x1 - 1
            bne x1, x2, _loop     // x2 = 0
    ```

Desdobrando o laço

- Assumindo as latências:

Instrução Produzindo	Instrução Usando	Latência
ALU	ALU	3 ciclos
ALU	Store	2 ciclos
Load	ALU	1 ciclo
Load	Store	0

Desdobrando o laço

- Sem qualquer escalonamento do código:

Instrução	ciclo
ld f0, 0(x1)	1
stall	2
add f4, f0, f2	3
stall	4
stall	5
sw f4, 0(x1)	6
add x1, x1, -8	7
bne x1, x2, _loop	8

- Escalonando o código temos:

Instrução	ciclo
ld f0, 0(x1)	1
add x1, x1, -8	2
add f4, f0, f2	3
stall	4
stall	5
sw f4, 0(x1)	6
bne x1, x2, _loop	7
,	

Desdobrando o laço

- Um esquema simples para aumentar o número de instruções em relação às instruções de desvio e overhead é o desdobramento (desenrolamento) do laço. O desdobramento simplesmente replica o corpo do laço várias vezes, ajustando o código de término do laço.

Desdobrando o laço

Desdobrando o laço 4 vezes:

```
_loop : ld    f0, 0(x1)
        add f4,  f0, f2
        sw   f4,  0(x1)
        ld     f6, -8(x1)
        add  f8,  f6, f2
        sw    f8, -8(x1)
        ld     f0, -16(x1)
        add  f12, f0, f2
        sw    f12, -16(x1)
        ld     f14, -24(x1)
        add  f16, f14, f2
        sw    f16, -24(x1)
        add  x1, x1, -32
        bne  x1, x2, _loop
```

Desdobrando o laço

Escalonando o código:

```
_loop : ld    f0, 0(x1)
        ld     f6, -8(x1)
        ld     f12, -16(x1)
        ld     f14, -24(x1)
        add f4,  f0, f2
        add  f8,  f6, f2
        add  f14, f12, f2
        add  f16, f14, f2
        sw   f4,  0(x1)
        sw    f8, -8(x1)
        sw    f14, -16(x1)
        sw    f16, -24(x1)
        add  x1, x1, -32
        bne  x1, x2, _loop
```

Desdobrando o laço

- O ganho com o escalonamento no laço desenrolado é ainda maior do que no laço original. Esse aumento ocorre porque o desenrolamento do laço expõe mais computação que pode ser escalonada para minimizar os stalls; o código anterior não apresenta stalls.
- Escalonar o laço dessa forma exige a compreensão de que as cargas e os armazenamentos são independentes e podem ser intercambiados.